多重選択ナップザック問題における計算量$O(n)$の検討(最適化の数理における離散と連続構造)

AUTHOR(S):

辻, 光宏; 仲川, 勇二

CITATION:

辻, 光宏 ...[et al]. 多重選択ナップザック問題における計算量$O(n)$の検討(最適化の数理における離散と連続構造). 数理解析研究所講究録 1996, 945: 121-129

ISSUE DATE:

1996-04

URL:

http://hdl.handle.net/2433/60217

数理解析研究所講究録
945巻 1996年 121-129

多重選択ナップザック問題における計算量$O(n)$の検討

関西大学総合情報　辻　光宏（Mitsuhiro Tsuji）
仲川勇二（Yuuji Nakagawa）

1．多重選択ナップザック問題での時間計算量

多重選択ナップザック(Multi-Choice Knapsack)問題をコンピュータを用いて解く場合に重要な役割を担うのは、最適値の限界値(Bound)の評価である。限界値を求める一般的な方法は、変数の整数条件を除いてできあがる線形計画問題すなわちＬＰ緩和問題を解くことであり、そのための各種の解法アルゴリズムが提案されている。

各手法の理論的に見積もられる時間計算量は、Sinha-Zoltners[1979]、Glover-Klingman[1979]、Ibaraki[1978]の解法では$O(n\log n)$である。一方、Zemel[1980]では$O(n\log n_{\max})$、さらにDyer[1978]では$O(n)$である。ここで、nは変数の総数、$n_{\max}$は同一クラスに属する変数個数の最大数である。時間計算量が$O(n\log n)$である各手法は、内部で利用するソートのア

ルゴリズム（クイックソート）の時間計算量に依存しているために生じる理論的な時間計算量である。$O(n\log n)$以外の解法はアルゴリズムの中でソート処理を回避する工夫により実現している。時間計算量が$O(n)$であり、クイックソートと比較しても実用的に遜色のないソート処理のアルゴリズムに対して、検証する。

2．ソート処理の時間計算量

キー比較に基づくソートのアルゴリズムには、クイックソート(quick sort)がある。クイックソートは、実用的に高速であるので非常によく使われるが、n個の配列に対して一般的に時間計算量 $O(n\log n)$で整列し、最悪の場合には時間計算量 $O(n^2)$で整列する事が広く知られている。

一方、対象データの離散性を生かし、基数(radix, base) を元にしたソートアルゴリズムが開発された。これは、n個の配列に対して、最悪の場合にでも 時間計算量 $O(n)$で整列できるという大きな長所を持っている。しかしながら、アルゴリズムの基本となるのが限定された整数であるために、一般的な実用面で欠点を持つ。

対象データの離散性を別の形で生かした番地計算(address calculation)を元にしたソート手法としてバケツソ

ート(bucket sort)がある。これは、番地計算整列法の理論面からの計算時間は、一様分布の配列に対しては$O(n)$であるが、特殊な例に対しては$O(n^2)$時間であると言われている。番地計算法を改良したソートアルゴリズムとして仲川ら[1981]によって提案されたRAC (Revised Address Calculation) ソートがある。これは、その後の改良により、計算量$O(n)$で整列でき、実用面でもクイックソートと遜色無いものとなった。

3．ＲＡＣソート

3.1　ＲＡＣソートの考え方

要素数がnの配列 $\{a_1, a_2, \cdots, a_n\}$を整列するものとする。再帰的にRACソートを行うかを決定するため、１つのRACソートが取り扱う最小要素数αをあらかじめ設定する。

・整列すべき要素数がα以上の場合 RACソートを再帰的に用い、その他の場合は他の整列法を用いる。１つの RACソートが扱う対象の配列は、$\{a_{i^{FST}}, a_{i^{FST}+1}, \cdots, a_{i^{LST}}\}$である。

・最大値a_{MAX}と最小値a_{MIN}を求める。

$$a_{MIN} = \min\{a_i \mid i = i^{FST}, i^{FST}+1, \cdots, i^{LST}\}$$
$$a_{MAX} = \max\{a_i \mid i = i^{FST}, i^{FST}+1, \cdots, i^{LST}\}$$

・配列$\{a_{i^{FST}}, a_{i^{FST}+1}, \cdots, a_{i^{LST}}\}$を分割する数$n^{DIV}$を次の式で決定する。　　$n^{DIV} = \tau(i^{LST} - i^{FST} + 1)$　　ここで、τ は、あらかじめ設定する 1.0 未満の正の数字である。

・範囲(a_{MIN}, a_{MAX})をn^{DIV} 個の等間隔の部分区画に分割し、要素a_i は、次式で与えられるサブグループ番号に属するようにする。

$$d(a_i) = \lfloor (n^{DIV} - 0.01)(a_i - a_{MIN}) / (a_{MAX} - a_{MIN}) \rfloor + 1$$

ここで、$\lfloor y \rfloor$ はyを越えない最大整数である。

以上の手続きを、サブグループが十分に小さくなるまで再帰的に繰り返して適用する。

3.2　ＲＡＣソートの再帰過程

```
DEFDATA
    PARA = {α, τ}
    ARRAY  = {i^FST, i^LST, {a_1, a_2, ···, a_n}}
    STACK  = {n^S, {s_1, s_2, ···, s_n^S}};
    A and S = {ARRAY , STACK }
    BUCKETS = {n^B, {b_1, b_2, ···, b_n^B }};
 ENDDEF

FUNCTION RACSort ()
  INPUT PARA, A and S ;
      n^DIV ← ⌊ τ (i^LST − i^FST + 1}⌋ ;
    {BUCKETS , ARRAY } ⇐ Partition(n^DIV , ARRAY ) ;
    {A and S} ⇐ Classification(α, BUCKETS , A and S ) ;
    IF n^S > 0 THEN
        {i^FST} ⇐ Pop(STACK ) ;
        {i^LST} ⇐ Pop(STACK ) ;
        {A and S} ⇐ RACSort (PARA , A and S) ;
    ENDIF
  OUTPUT A and S ;
ENDFUNC
```

図１：ＲＡＣソートの再帰過程

図１にＲＡＣソートの再帰過程アルゴリズムを示す。

DEFDATA と ENDDEF の間では，データの階層的な定義を行っている。C または C++では構造体、Pascal または Modula-2 ではレコード型を用いて実現できる。

記号⇐は関数の出力を意味する。たとえば、$\{A,B\} \Leftarrow Func(C,D)$は、入力データとしてデータ列$C,D$を用い、関数$Func$を実行し,データ列$A,B$を出力することを意味する。すなわち、この式によって、データ列A,Bの内容が書き換えられることになる。

関数*Partition* は，配列$\{a_1,a_2,\cdots,a_n\}$のi^{FST}からi^{LST}番目までの要素を、番地計算関数$d(a_i)$を使って計算したサブグループ番号を用いて、n^{DIV} 個のサブグループに分割する。
分割はサブグループ１からサブグループn^{DIV} までが順番になるよう*ARRAY* 内で要素を入れ替え、各サブグループの最初の位置は*BUCKETS* 内に記録する。この関数内では*BUCKETS* は作業領域節約のためにカウンターとしても利用する。この関数の出力は A and Sである。

関数*Classification* は、要素数がα 以上の場合は、そのサブグループの*ARRAY* 内での最初と最後の位置を*STACK* に積み上げる（*Push*)。

4．ＲＡＣソートの時間計算量$O(n)$

RAC ソートは、kビットn個の実数値要素からなる配列に対して次のような性質をもつ。

［性質１］

再帰の深さ（関数が呼び出された回数）p の最大値p^{UB} は $p^{UB}=\lceil k / \log_2(\tau\alpha)\rceil$ である。

（証明）

常に分割数n^{DIV} は$\tau\alpha$ 以上である。このため、分割により得られる情報量がk ビットを越えるような再帰の深さpは、次式で表すことができる。

$\log_2((\tau\alpha)^p)\geq k$　　これより、次式が得られる。

$p\leq k / \log_2(\tau\alpha)$　　（証明終わり）

性質 1 を用いた実際的な例として、$\tau=0.5$, $\alpha=1000$の場合を考える。32 ビット浮動小数点のとき、$p^{UB}=4$で、64 ビット浮動小数点のとき、$p^{UB}=8$である。これらは実用的な数値である。

［性質２］

再帰の深さが $p(\leq p^{UB})$であるグループに対して、関数 *Partition* が各グループを分割して得られるサブグループの数の総合計は τn を越えることはない.

（証明）

再帰の深さが$p-1$のときに生成されたサブグループで、空でないグループの要素の総数はnを越えないことから明らかである．　　（証明終わり）

再帰の深さpにおいて、関数*Partition*が必要とする計算時間の総合計は性質2よりτnに比例した時間である。また、要素数α未満のグループの個数はnを越えることはないので、関数*Classification*において整列に必要な合計の計算時間は$c_1 n$を上限値として持つ。ただし、c_1はα個の要素を整列するのに必要な計算時間の上限値である。したがって、再帰の深さpにおいて関数*Partition*と関数*Classification*が使用する計算時間T_pに対して、

$T_p < c_2 n$となる上限値c_2が存在する。

［性質1］よりRACソートでn個の要素の整列に必要な計算時間Tは次式で表すことができる。

$$T = \sum_{p=1}^{\nu} T_p < p^{UB} c_2 n < ckn$$

ただし、cは適当な定数である。kビットn個の実数値配列を最悪の場合でも$O(n)$で整列することがわかる。

５．ＲＡＣソートの実用性

３２ビット実数データに対するクイックソートと基数ソ

ート（ラディクスソート）とＲＡＣソートそれぞれの時間計測結果を以下に示す。

表１　32ビット実数の実行時間（秒）比較表(NWS185)

要素数	3000	10000	30000	100000	300000
クイック	0.098	0.372	1.25	4.67	15.2
基数	0.266	0.878	2.70	9.21	27.8
Ｒａｃ	0.140	0.484	1.53	5.34	16.3

表１の結果より、クイックソートと基数ソートを比較すると、基数法は最悪の場合でも $O(n)$ の整列法ではあるが、非現実的なほど大きな要素数にならないかぎり、基数法はクイックソートには勝てないことがわかる。RAC ソートの場合は，データの分布にも多少依存するが、表 1 からの概算で $n=5.3\times10^6$程度の要素数の時にクイックソートよりも速くなることが予想される。

７．参考文献

P.Sinha, A.A.Zoltners(1979). The multiple-choice knapsack problem. Operations Research 27, 503-515.

F.Glover, D.Klingman(1979). A $O(n\log n)$ algorithm for LP knapsacks with GUB constraints. Mathematical Programming 17, 345-361

T.Ibaraki, T.Hasegawa, K.Teranaka, J.Iwase(1978). The multiple choice knapsack problem, Journal of Operations Research Japan, 21, 59-95

E.Zemel(1980). The linear multiple choice knapsack problem, Operations Research, 28, 1412-1423

M.E.Dyer(1978). An algorithm for the multiple-choice knapsack linear program, Mathematical Programming, 29, 57-63

仲川，疋田(1981). 改訂番地計算分類法,電子情報通信学会論文誌, J64-D, 8, 737-741